SUBMITTED TO: 11TH SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS

(2)

Assigning Sites to Redundant Clusters in a Distributed Storage System*

Antoine N. Mourad
W. Kent Fuchs
Daniel G. Saab

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

Contact: Antoine N. Mourad
mourad@crhc.uiuc.edu
Ph. (217) 244-7180
Fax (217) 244-5686

Abstract

Distributed redundant disk arrays can be used in a distributed computing system or database system to provide recovery in the presence of disk crashes and temporary and permanent failures of single sites. In this paper, we look at the problem of partitioning the sites into redundant arrays in such a way that the communication costs for maintaining the parity information are minimized. We show that the partitioning problem is NP-hard and we propose several heuristic algorithms for finding approximate solutions.

*This research was supported in part by the National Aeronautics and Space Administration (NASA) under Contract NAG 1-613 and in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Grant N00014-91-J-1283.

1 Introduction

Redundant disk arrays are used for the purpose of providing reliable storage while increasing the I/O bandwidth in high performance systems [1, 2]. Redundant disk arrays can also be used in a distributed setting to increase availability in the presence of temporary site failures, disk failures, or major disasters [3]. In this environment, they are used as an alternative to multicopy schemes which are much more costly in terms of storage requirements. Cabrera and Long [4] have proposed the use of redundant distributed disk striping in a high speed local area network to support such I/O intensive applications as scientific visualization, image processing, and recording and play-back of color video.

When Distributed Redundant Disk Arrays (DRDA) are used, sites are grouped together to form a redundant array containing data and parity and capable of recovering from a single site failure. The size of each array is fixed and is determined by the tradeoff between the availability requirements of the system and the cost of the storage overhead. Hence a large distributed data storage system may have to be divided in several arrays of fixed size. In this paper we look at the problem of partitioning the distributed storage systems into fixed size arrays in such a way as to minimize the cost of remote accesses that have to be performed to update the parity information. This problem is somewhat related to the problem of file allocation and replica placement in a distributed system which has been studied extensively in the literature [5, 6]. However the two problems are different in nature because, in the DRDA case, there is one redundant item for N data items while in the file allocation problem each file is replicated several times. More importantly in the replica placement problem there

is no stringent constraint on the number of sites "sharing" a replica because when the replica becomes unavailable those sites can access the second nearest replica while in the DRDA case there is a hard constraint on the number of sites in an array.

In the following section, we describe a typical redundant disk array organization. In Section 3, we present the model used to formulate the problem mathematically and we prove that the problem is NP-hard. In Section 4, heuristic algorithms for solving the problem are described and results from an experimental evaluation are presented. In Section 5 we develop heuristics with guaranteed bounds on the deviation from the optimal cost. Section 6 deals with the problem of hot spots and with non-uniform site capacity. Finally, Section 7 contains some conclusions.

2 Distributed Redundant Disk Array Organization

Stonebraker and Schloss [3] proposed the DRDA organization shown in Figure 1. The data at each site is partitioned into blocks and data blocks from different sites are grouped into a parity group. The bitwise parity of the data blocks in each parity group is computed and written at a different site. In Figure 1, S_i denotes site i, D_{ij} denotes a data block, P_i denotes a parity block and S_i denotes a spare block, all at site i. The number under *block* in the first column of the figure denotes the physical block number on disk. Each row in the figure represents a parity group. The position of the parity block is rotated among the sites in order to avoid creating a bottleneck at the site where parity is stored. For every update to one of the data blocks in the parity group, the parity block needs to be updated using the

block	$Site_0$	$Site_1$	$Site_2$	$Site_3$	$Site_4$	$Site_5$
0	P_0	S_1	D_{20}	D_{30}	D_{40}	D_{50}
1	D_{00}	P_1	S_2	D_{31}	D_{41}	D_{51}
2	D_{01}	D_{10}	P_2	S_3	D_{42}	D_{52}
3	D_{02}	D_{11}	D_{21}	P_3	S_4	D_{53}
4	D_{03}	D_{12}	D_{22}	D_{32}	P_4	S_5
5	S_0	D_{13}	D_{23}	D_{33}	D_{43}	P_5

Figure 1: Organization of a distributed redundant disk array ($N = 6$).

following formula:

$$P_{new} = (D_{old} \oplus D_{new}) \oplus P_{old}.$$

Spare blocks are provided in order to be able to reconstruct data blocks that become inaccessible due to a site failure. The failed data block is reconstructed by XORing all other data blocks and the parity block in its parity group. If K denotes the number of data blocks per parity group then $N = K + 2$ denotes the number of sites in a distributed disk array. The storage overhead for the parity and spare blocks required by DRDAs is $(200/K)\%$ compared to a 100% overhead for the case of two copy schemes.

3 The Model

We model the distributed computing system by an undirected connected graph $G = (V, E)$ where V is the set of sites and each edge $e \in E$ represents a bidirectional communication link between two sites. For each $e \in E$, w_e denotes the cost of communication over link e *. We assume that if n is the number of sites in V then $n = mN$ for some m. We will assume that the site capacity is uniform. In Section 6.2 we show how to deal with non-uniform site

*For $e = (u, v)$, w_e could be the actual distance between site u and site v.

capacity. In the pattern shown in Figure 1, the parity blocks of the $N-2$ data blocks of site i reside on sites $i+1$ mod N through $i+N-2$ mod N. Therefore there is no parity update traffic from site i to site $i+N-1$ mod N. In order to make the problem symmetrical and thus easier to tackle, we assume that the pattern shown in Figure 1 is rotated for the next set of N blocks so that update traffic from a given site is distributed over the remaining $N-1$ sites. This will also provide more load balancing for the parity update traffic. Let μ_v designate the rate of update accesses to data blocks at site v. Each update will cause communication between the site where the update took place and the site holding the parity for the given data block. At each site the set of data blocks that have their corresponding parity blocks on the same site is called a data group. To simplify the model, we assume that the $N-1$ data groups share equally the update rate. This implies that the rate at which site v sends parity update information to each other site in its redundant array is $\lambda_v = \mu_v/(N-1)$. This assumption is supported by the fact that consecutive data blocks have their parity blocks on different sites which implies that accesses to a heavily used file that is stored on consecutive disk blocks will be spread over different data groups. In Section 6, the above assumption will be removed. The problem of partitioning the sites into arrays of size N in such a way that parity update costs are minimized can be mathematically formulated as follows:

Problem 1 (SP) *Find a partition of V into m disjoint subsets $V_1, V_2, \ldots, V_m$ of size N such that if $d(u,v)$ denotes the length of the shortest path between u and v then $\sum_{i=1}^{m} \sum_{u \in V_i} \lambda_u \sum_{v \in V_i - \{u\}} d(u,v)$ is minimum.*

Theorem 1 *Problem SP is NP-hard for any fixed $N \geq 3$.*

Proof: We prove that problem SP is NP-hard by showing that there is a polynomial time transformation from the problem of partitioning a graph into cliques of size N to problem SP. The Partition into Cliques of size N (PC) problem can be stated as follows:

Instance: A graph $G = (V, E)$, with $|V| = Nm$ for some positive integer m.

Problem: Is there a partition of V into m disjoint subsets V_1, V_2, ..., V_m such that, the subgraph of G induced by V_i is a clique of size N (complete graph with N nodes)?

PC is NP-complete for any fixed $N \geq 3$ (see Partition into Isomorphic Subgraphs [7]). To transform an instance of PC into an instance of SP, it is sufficient to set $\lambda_v = 1$ for all $v \in V$, and $w_e = 1$ for all $e \in E$. Then graph G can be partitioned into cliques of size N if and only if the cost of the optimal solution to the above instance of problem SP is $n(N-1)$.

□

The cost function $\sum_{i=1}^{m} \sum_{u \in V_i} \lambda_u \sum_{v \in V_i - \{u\}} d(u,v)$ can be rewritten as $\sum_{i=1}^{m} \sum_{u,v \in V_i, u \neq v} (\lambda_u + \lambda_v) d(u,v)$ $= \sum_{i=1}^{m} \sum_{u,v \in V_i, u \neq v} D(u,v)$, where $D(u,v)$ is defined as $D(u,v) = (\lambda_u + \lambda_v) d(u,v)$. In this form the general problem is reduced to a uniform load problem with the distance D replacing d. However D is not a true distance since it does not necessarily satisfy the triangular inequality.

4 Approximation Algorithms

4.1 Description of the Heuristics

The first heuristic is based on a greedy strategy that consists of satisfying first the sites with the largest update rate. Let Λ be the list of update rates for all sites. When sites are grouped

into clusters their update rates are removed from Λ and replaced by a single update rate for the cluster. The cluster update rate is the total update rate of the sites in the cluster minus the fraction corresponding to parity update requests serviced within the cluster. If i sites with update rates $\lambda_1, \ldots, \lambda_i$ form a cluster, the update rate assigned to that cluster in Λ will be $(\sum_{k=1}^{i} \lambda_k)(1 - \frac{i-1}{N-1})$.

Algorithm 1:

Step 1. Select the largest value in Λ and let a be the corresponding site (or cluster). Find the site (or cluster) b such that merging a and b results in the smallest increase in the cost function. Merge the two sites (or clusters) if the resulting cluster has less than N sites and the total number of clusters does not exceed m. If the clusters cannot be merged, find the next best choice for b and repeat. Remove the update rates of the merged sites (or clusters) from Λ and replace them with the cluster update rate computed as shown above.

Step 2. Repeat Step 1 until m clusters having N sites each have been formed.

The computational cost of Algorithm 1 is $O(Nn^2)$. But it requires that the all-pair shortest path algorithm be performed first which requires $O(n^3)$ operations.

The second approach consists of two stages: in the first stage m sites are identified to be used as cluster seeds and in the second stage the remaining sites are allocated to the clusters to form m subsets of N sites each.

Algorithm 2:

Step 1. Select the two sites with the largest distance between each other and include them in the set S of cluster seeds.

Step 2. Select the site v with the largest average distance to the sites already in S and add it to S.

Step 3. Repeat Step 2 above until $|S| = m$. Each cluster initially contains one of the m seeds in S.

Step 4. For each of the m clusters, compute the average update rate of the sites in the cluster. In decreasing order of their average update rate, allocate to each cluster the site that is closest to it in terms of the distance metric D.

Step 5. Repeat Step 4 above until all sites have been allocated to the m clusters.

We use the distance metric D in Step 4 because it provides the actual increase in the cost function of a cluster when a node is added to it. The computational cost of the Algorithm 2 is $O(Nn^2)$. It also requires that the all-pair shortest path algorithm be performed first.

The third approach is based on the hierarchical clustering technique [8]. We use the distance matrix whose entries are $d(u, v)$ for all $u, v \in V$. Clusters are formed by merging together sites or smaller clusters that are close to each other. When two sites (or clusters) are grouped together, the distance matrix is modified by eliminating the columns and rows corresponding to the merged sites (or clusters) and replacing them with a single column and a single row reflecting the average distance between the merged sites and other sites (or clusters). The procedure is as follows:

Algorithm 3:

Step 1. Find the smallest entry in the distance matrix and merge the two sites (or clusters) together if the resulting cluster has N sites or less and if the total number of clusters does

not exceed m. If any of the latter conditions is not satisfied, select the next smallest entry and repeat. Once two sites (or clusters) have been merged, update the distance matrix and the number of clusters accordingly.

Step 2. Repeat Step 1 above until m clusters having N sites each have been formed.

The complexity of Algorithm 3 is $O(n^3)$. When an initial partition has been found using Algorithm 1, 2 or 3, the following procedure can be used to improve it.

Procedure Improve:

Step 1. Select the site u with the highest update rate. For each site v outside site u's partition, compute the change in cost $\Delta C(u,v)$ if u and v were swapped. Let v^* be the site corresponding to the minimum change in cost: $\Delta C(u,v^*) = \min \Delta C(u,v)$. If $\Delta C(u,v^*) < 0$ then swap u and v.

Step 2. Repeat Step 1 above for all sites in V in decreasing order of their update rate.

The complexity of the above procedure is $O(n^3)$. The procedure can be repeated several times as long as it improves the total cost.

4.2 Experimental Evaluation

We have conducted experiments to evaluate the approximate solutions obtained using the heuristics and to compare the three proposed approaches for site assignment. In the experiments, we used randomly generated graphs. The distance on each edge in the graph was drawn from a uniform distribution over the interval $[1,K_w]$. The update rates at each site were drawn from a uniform distribution over the interval $[1,K_\lambda]$.

Table 1: Comparison between approximate solutions and the optimal solution.

K_w, K_λ	Random	Algorithm 1	Algorithm 2	Algorithm 3	Exhaustive
1000, 10	68400	52439	53462	52649	47475
100, 100	66071	50012	51347	51237	45661
10, 1000	96757	76388	77362	77062	70004

In our experiments we found out that Algorithm 2 performs better when the distance D is also used in the first stage of the algorithm. This can be explained by the fact that using D in the generation of the cluster seeds ensures that edges with large $D(u, v)$ will not be used within a cluster, i.e., sites that have large loads and that are far apart are not placed in the same cluster. The results shown here for Algorithm 2 were obtained using D instead of d.

In the first experiment, we compare the approximate solution provided by the heuristics to the optimal solution. The optimal solution was obtained using exhaustive search. N was taken to be equal to 5 and n equal to 15. Table 1 shows the results for three situations: one where the edge weights vary more widely than the site loads, one where both are picked from the same interval and one where the site loads vary more widely than the edge weights. Each entry represents the average over 100 randomly generated graphs. The costs of the approximate solutions are within 10% of the cost of the optimal solution. In the first column of the table, we have listed the cost of a random solution.

Since, in the first experiment, an exhaustive search was used to find the optimal solution, the number of nodes n could not be very large. In a second experiment, we compared the performance of the three heuristics for larger values of n. N was set to 10. Figure 2

shows the results for the second experiment. For clarity of the figure, we plotted the cost of the approximate solution divided by 1000. We can see from the figure that Algorithm 3 outperforms Algorithms 1 and 2 for all values of n except when $n = 20$ in which case Algorithm 2 performs better. For the first and second environments Algorithm 1 outperforms Algorithm 2 for large values of n but for the last environment Algorithm 2 outperforms Algorithm 1. The main point that can be deduced from this experiment is that, in spite of the fact that Algorithm 3 does not use any information about site loads, it outperforms the other two algorithms for large n. This means that, for large n, it is more important to minimize the sum of the edge weights within each cluster than to use the greedy approach that attempts to assign to the sites with large loads their nearest neighbors.

5 Heuristics with Performance Guarantees

The heuristics described in Section 4 provide in general a good approximate solution. However, there is no guarantee that the approximate solution will not diverge significantly from the optimal one in certain cases. In this section, we seek to find a heuristic for which it is possible to establish a bound on the error between the approximate solution and the optimal one. We develop such a heuristic first for the case of a system with balanced load, $\lambda_v = \lambda$, for all $v \in V$, and uniform edge weights, then we look at the more general case of a balanced load system with arbitrary edge weights. Since a problem with arbitrary site loads can always be transformed into a problem with uniform site load as shown in Section 3, then the heuristic for the balanced load case with arbitrary edge weights will also provide

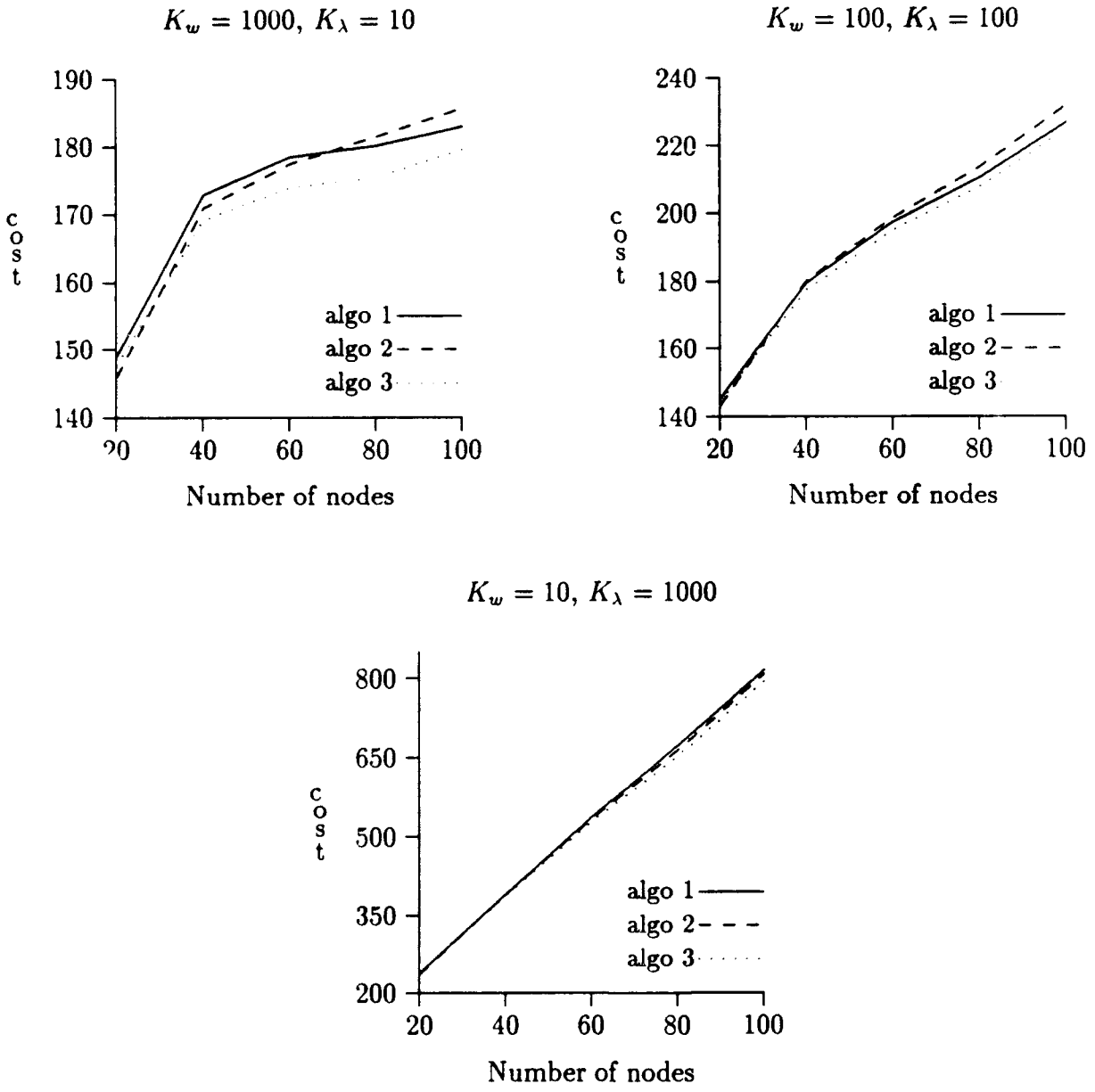

Figure 2: Comparison between the three heuristics.

performance guarantees for the arbitrary load case.

5.1 Balanced Load and Uniform Edge Weights

The heuristic requires the use of a spanning tree with many leaves. The problem of finding a spanning tree with a maximum number of leaves is NP-hard [7] however there exist polynomial time algorithms for generating spanning trees with many leaves. Typically these methods guarantee that a certain fraction of the nodes will be leaves. The fraction of leaves is a function of the minimum degree k of the graph. Kleitman and West proved the following result [9]:

Theorem 2 (Kleitman-West) *If k is sufficiently large, then there is an algorithm that constructs a spanning tree with at least $(1 - b\ln k/k)n$ leaves in any graph with minimum degree k, where b is any constant exceeding 2.5.*

It was also conjectured that a spanning tree can be constructed with a larger fraction of leaves. More specifically, Linial conjectured that the number of leaves could be at least $\frac{k-2}{k+1}n + c_k$. This stronger result was proved for $k = 3$ with $c_3 = 2$ and for $k = 4$ with $c_4 = 8/5$ [9].

Algorithm

Step 1. Find a spanning tree with many leaves.

Step 2. Partition the spanning tree into m clusters of N nodes each using procedure **Partition_Tree** described below.

The partition found for the tree will be used for the original graph. In the description of the procedure **Partition_Tree**, we assume that the tree is levelized starting from the root.

Procedure Partition_Tree:

The procedure partitions the tree from the bottom up. As the clusters are built, whenever the size of a cluster reaches N nodes, that cluster is removed from the tree. Starting from the deepest level in the tree, sibling leaves are placed together in a cluster. If all siblings have been used then their parent is included in the cluster. At an internal node v, all subtrees rooted at its siblings must be processed so that only less than N nodes are left in each subtree. Those subtrees are numbered from 1 to $d(v) - 1$, $d(v)$ being the degree of v. Then the clusters are formed by adding to the nodes of subtree i enough nodes from subtree $i + 1$ to make an N node cluster. If there are not enough nodes in subtree $i+1$ to form a complete cluster, the nodes of the two subtrees are placed together and the next subtree is used to complete the cluster. If all the subtrees have been used, and there remains an incomplete cluster then the parent node is added to the remaining cluster and the procedure continues at the next level. When adding a portion of the nodes of a given subtree to the preceding subtree(s) to complete a cluster, the nodes at the deepest level in that subtree are used first so that removal of the newly completed cluster will not disconnect the tree.

Theorem 3 *The cost (HEU) of the approximate solution found using a spanning tree with many leaves and the cost (OPT) of the optimal solution satisfy the following relationship:*

$$\frac{\text{HEU}}{\text{OPT}} \leq 2\alpha + (1 - \alpha)\frac{N^2}{N - 1},$$

where α is the fraction of leaves in the spanning tree.

Proof We need to establish an upper bound on the cost of the approximate solution and a lower bound on that of the optimal one. The cost in the graph of the approximate solution is at most the cost of that solution in the tree. We evaluate the cost in the tree by adding up the contributions of each edge in the spanning tree to the overall cost. If an edge connects a leaf node to the tree it will be referred to as a leaf edge otherwise it will be called an internal edge. A leaf edge will be used in only one cluster and it will be used only for communication between the leaf node and the other $(N-1)$ nodes in the cluster. Therefore the contribution of a leaf edge to the overall cost is $2(N-1)$. An internal edge will be used in at most two clusters and in each cluster it will be used by i nodes to communicate with the other $N-i$ nodes in the cluster. If α designates the fraction of leaf nodes in the tree, we have:

$$\begin{aligned} \text{HEU} &\leq \alpha n \times 2(N-1) + (n-1-\alpha n) \times 2 \times \max_{1 \leq i \leq N-1} 2i(N-i) \\ &\leq n(N-1)(2\alpha + (1-\alpha)N^2/(N-1)) \end{aligned}$$

For the cost of the optimal solution, an obvious lower bound is the cost in a complete graph which is $n(N-1)$. Hence $\text{HEU/OPT} \leq 2\alpha + (1-\alpha)N^2/(N-1)$. □

As stated in Theorem 2, for large k, α converges to 1 and the above bound approaches 2. Note that it is reasonable to assume that the minimum degree will be large in practice because the underlying network has to have sufficient connectivity to enable communication under node failures and hence has to have a reasonably large minimum degree.

The complexity of the algorithms for generating trees with many leaves [9] is $O(|E|)$. The complexity of the **Partition_Tree** procedure is $O(n)$.

5.2 Balanced Load and Arbitrary Edge Weights

For arbitrary edge weights the problem of finding a heuristic with guaranteed performance bounds is much harder. In the following we describe a heuristic for which a worst case performance bound can be established. The bound is more significant for systems where link communication costs (edge weights) do not vary widely. The heuristic consists of finding a minimum spanning tree, partitioning the tree into clusters using procedure **Partition_Tree** and using that partition as an approximate solution. The following result will be used to establish a lower bound on the cost of the optimal solution.

Lemma 1 *In a complete graph, the average weight of the edges in a minimum spanning tree is at most the average weight of all edges.*

Proof We use induction on the number of nodes n. The lemma is obviously true for $n = 2$ or $n = 3$. Suppose it is true for graphs with $n - 1$ nodes and consider an n-node graph. Select node v such that the average weight of edges incident on v is at least the average weight of all edges in the graph. Remove v from the graph and find a minimum spanning tree in the remaining $(n-1)$-node graph. Then add to this spanning tree the lightest edge e^* connecting v to the other nodes to form an n-node spanning tree. Let MST_{n-1} and MST_n be the total weight of the $(n-1)$-node and the n-node spanning trees respectively. Let $\mathcal{E}(v)$ be the set of edges incident on v. Using the induction hypothesis, we have:

$$\begin{aligned}
\mathrm{MST}_n &\leq \mathrm{MST}_{n-1} + w_{e^*} \leq \frac{\sum_{e \in E - \mathcal{E}(v)} w_e}{(n-1)/2} + \frac{\sum_{e \in \mathcal{E}(v)} w_e}{n-1} \\
&\leq \frac{\sum_{e \in E - \mathcal{E}(v)} w_e}{(n-1)/2} + \frac{\sum_{e \in \mathcal{E}(v)} w_e}{n-1} + \frac{\sum_{e \in \mathcal{E}(v)} w_e}{n-1} - \frac{\sum_{e \in E} w_e}{n(n-1)/2}
\end{aligned}$$

$$= \frac{\sum_{e \in E} w_e}{n/2}.$$

□

To obtain a lower bound on the cost of the optimal solution, we consider the optimal partition and we build a spanning tree by first finding a minimum spanning tree in each cluster and then replacing each cluster by a single node and connecting each pair of these nodes by the lightest edge linking the initial clusters. An intercluster minimum spanning tree is then found. The intracluster spanning trees along with the intercluster spanning trees form a spanning tree for the entire graph.

Lemma 2 *The list of edge weights of the intercluster minimum spanning tree (ICMST) is included in the list of edge weights of the global minimum spanning tree (GMST).*

Proof Let e be an edge in the ICMST that does not appear in the GMST. Let u and v be its endpoints in the original graph and let w be its weight. The path in the GMST from u to v induces a path in the intercluster graph from the cluster of u to that of v. If the path is a single edge then this edge must have weight w and could replace the edge e in the ICMST. If the induced path has more than one edge then, since the ICMST cannot contain a cycle, some of the edges on the induced path must not appear in the ICMST and at least one of these induced edges that do not appear in the ICMST forms a cycle containing e when added to the ICMST. Let e' be such an edge. e' must have weight at most w otherwise it could be replaced in the GMST by (u, v) to obtain a spanning tree with a smaller cost. In addition e' cannot have weight less than w because it would then be possible to replace e by e' in the

ICMST and obtain a smaller intercluster spanning tree. Hence the weight of e' is w and we could remove e and replace it with e' in the ICMST. This process can be repeated until all edges in the ICMST also appear in the GMST. Hence the lemma is proved. □

Theorem 4 *The cost (HEU) of the approximate solution found using a minimum spanning tree and the cost (OPT) of the optimal solution satisfy the following relationship:*

$$\frac{\text{HEU}}{\text{OPT}} \leq N \frac{\text{MST}}{\text{MST} - (m-1)\overline{w}},$$

where MST is the total weight of the edges in the minimum spanning tree and $\overline{w}$ is the average weight of the $m-1$ heaviest edges in the minimum spanning tree.

Proof In evaluating an upper bound on the cost of the approximate solution, we follow the same procedure as in the proof of Theorem 3 but we will not distinguish between leaf edges and internal edges. Each edge e in the tree will be used by at most two clusters and the contribution of e to the overall cost is bounded by $2 \times w_e \times \max_{1 \leq i \leq N-1} 2i(N-i)$. Hence we have $\text{HEU} \leq N^2\text{MST}$.

Let MST_i be the weight of the minimum spanning tree of cluster i for $1 \leq i \leq m$ and MST_c be the weight of the intercluster tree. We have $\sum_{i=1}^{m} \text{MST}_i + \text{MST}_c \geq \text{MST}$. Using Lemma 2, we have $\sum_{i=1}^{m} \text{MST}_i + (m-1)\overline{w} \geq \text{MST}$. Let OPT_i be the contribution to the optimal cost by cluster i. Using Lemma 1 we have $\text{OPT}_i/N \geq \text{MST}_i$ therefore $\text{OPT} \geq N(\text{MST} - (m-1)\overline{w})$.

□

Let r be the ratio of the largest edge weight to the smallest edge weight. A looser but simpler bound than the one established in Theorem 4 can be derived using the parameter r:

$$\text{HEU/OPT} \leq N\left(1 + \frac{m-1}{n-m}r\right) \leq N(1 + r/(N-1)).$$

6 Generalization of the Model

6.1 Non-Uniform Load within Site

In our model, we assumed that each site sends parity updates to each other site in its partition at the same rate. This implies a uniform update rate to each of the $N-1$ data groups of a given site that have parity information on each of the $N-1$ other sites. If the update rate information for each data group at each site is available then the model can be refined to account for the difference in the rate of parity update requests issued by a given site and destined to the other sites in the array. The refined model should yield better results in the presence of hot spots. The update rate λ_u of site u is replaced by $N-1$ update rates $\lambda_{u,1}$, ..., $\lambda_{u,N-1}$ corresponding to each of its data groups. In this case, an obvious optimization would be to have the parity of the i^{th} most frequently accessed data group of a given site placed on the i^{th} nearest site in its partition. Note that this can be implemented without having to reshuffle the data on disk by saving the permutation describing the remapping of the $N-1$ data groups for each site and using it to send parity update requests to the proper site. Given this optimization, a greedy strategy for solving the partitioning problem is described in the following.

Algorithm Greedy

Let Λ be the list of update rates for all data groups at all sites.

Let p_v be the number of site v's partition. Initially $p_v = -1$ for all $v \in V$.

Let n_i be the number of sites in partition i. Initially, $n_i = 0$. Assume $n_{-1} = 0$ throughout.

Let k be the current number of partitions. Initially $k = 0$.

Let $\mathcal{N}(v) = V - v$, for all $v \in V$.

Let $l = 0$.

Step 1. Select the largest value λ in Λ and let u be the corresponding site. If $n_{p_u} = N$ go to Step 5.

Step 2. Find the nearest site to u in $\mathcal{N}(u)$. Call it v. If p_u or $p_v \neq -1$ and $n_{p_u} + n_{p_v} < N$ or if $p_u = p_v = -1$ and $k < m$ go to Step 4.

Step 3. Remove v from $\mathcal{N}(u)$. If $\mathcal{N}(u) = \emptyset$ go to Step 5, otherwise go to Step 2.

Step 4. If $p_u = p_v = -1$ set $p_u = p_v = l$, $n_l = 2$, $l = l + 1$, and $k = k + 1$.

If $p_u = -1$ and $p_v \neq -1$ set $p_u = p_v$ and $n_{p_v} = n_{p_v} + 1$.

If $p_u \neq -1$ and $p_v = -1$ set $p_v = p_u$ and $n_{p_u} = n_{p_u} + 1$.

If $p_u \neq -1$ and $p_v \neq -1$, set the partition number for every site in v's current partition to p_u, set $n_{p_u} = n_{p_u} + n_{p_v}$, $n_{p_v} = 0$, and $k = k - 1$.

Step 5. Remove λ from Λ. If $\Lambda = \emptyset$ stop.

Step 6. If $\sum_i n_i < n$, go to Step 1, otherwise stop.

The algorithm uses the same basic idea as Algorithm 1 of Section 4. It tries to satisfy first the nodes with highest update rate. Its complexity is $O(Nn^2)$ but as in the case of

Algorithm 1, it requires the all-pair shortest path algorithm.

6.2 Non-Uniform Site Capacity

The case of non-uniform site capacity can be handled in the same fashion as proposed by Stonebraker and Schloss [3]. We assume that the total number of disks is Np for some p[†] and that the number of disks at any given site is at most p. The system could then be partitioned using the following procedure.

Step 1. Select the $N\lfloor |V|/N \rfloor$ sites with the largest number of disks and apply one of the partitioning algorithms described in the previous sections to assign one disk from each of the selected sites to an array.

Step 2. Remove the assigned disks and remove sites with no disks left.

Step 3. Repeat the above steps until all disks have been assigned.

Non-uniform disk capacity can be dealt with by using logical disks of size B blocks such that the site capacities are multiples of B [3].

7 Conclusion

We looked at the problem of partitioning the sites of a distributed storage system into redundant disk arrays while minimizing the communication costs for updating the parity information. We showed that the problem is NP-hard in its general form. We investigated heuristic methods for obtaining approximate solutions to the site partitioning problem. We

[†]This replaces the assumption that $|V| = mN$.

also established guaranteed upper bounds on the deviation from the optimal cost for some of the heuristics. Future research includes evaluating different strategies for dealing with hot spots and solving the partitioning problem optimally for specific topologies.

References

[1] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the ACM SIGMOD Conference*, pp. 109–116, June 1988.

[2] J. Gray, B. Horst, and M. Walker, "Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput," in *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 148–161, Aug. 1990.

[3] M. Stonebraker and G. A. Schloss, "Distributed RAID – a new multiple copy algorithm," in *Proceedings of the Sixth IEEE International Conference on Data Engineering*, pp. 430–437, Feb. 1990.

[4] L.-F. Cabrera and D. D. E. Long, "Swift: Using distributed disk striping to provide high I/O data rates," Tech. Rep. UCSC-CRL-91-46, Computer and Information Sciences, University of California, Santa Cruz, 1991.

[5] L. W. Dowdy and D. V. Foster, "Comparative models of the file assignment problem," *ACM Computing Surveys*, vol. 14, pp. 287–313, June 1982.

[6] B. W. Wah, "File placement on distributed computer systems," *IEEE Computer*, pp. 23–32, Jan. 1984.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness.* New York: Freeman, 1979.

[8] M. R. Anderberg, *Cluster Analysis for Applications.* New York: Academic Press, 1973.

[9] D. J. Kleitman and D. B. West, "Spanning trees with many leaves," *SIAM Journal on Discrete Mathematics*, vol. 4, Feb. 1991.